Analysing FallingFruit.org's Data for Association Rules

```
# imports
#################################################################
# data management
import pandas as pd
import numpy as np
# data processing
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import fpgrowth, association_rules
# maps and graphs
import cartopy.crs as ccrs
import networkx as nx
from matplotlib import pyplot as plt
%matplotlib inline
#################################################################
```

Let's import the data and have a look

```
locations = pd.read_csv('data/0A_locations.csv')
types = pd.read_csv('data/0B_types.csv')
```

```
/var/folders/xx/1jz4bzvs0kdghy416zlmdn5c0000gn/T/ipykernel_86879/1964324172.py:1: DtypeWarning: Columns (1,6,7,8,10,13,15) have mixed types. Specify dtype option on import or set low_memory=False.
  locations = pd.read_csv('data/0A_locations.csv')
```

From the message above, we can see that some columns in `locations` have mixed types.

```
locations.head()
```

	id	type_ids	lat	lng	unverified	description	season_start	season_stop	no_season
0	22	3	37.409849	-122.137529	False	Nice big tart oranges	December	NaN	False
1	23	8	37.412087	-122.140182	False	Huge trees, many of them, all along the bike p...	NaN	NaN	False
2	24	4	37.412043	-122.139700	False	NaN	NaN	NaN	False
3	25	3	37.411562	-122.139288	False	NaN	NaN	NaN	False
4	26	4	37.411252	-122.138862	False	NaN	NaN	NaN	False

There are some `NaN` values in the data. At first glance a lot of these are in columns we won't need, but we can't take that for granted. Joining the table and filtering the columns we need will give us a better idea of the state of the data.

```
# Joining the data on id / type_ids will give us records where the type.id matches reco
# locations.type_ids columns. We should expect to see the same number of records in
# the joined table as we do in the locations table
initial_joined = pd.merge(types, locations, how='inner', left_on='id', right_on='type_i
```

```
print(locations.shape[0], initial_joined.shape[0])
```

```
1480141 458752
```

Looks like we've lost 66% of our data in the join. `types.id` is a primary key, so much be unique. The issue might lie in `locations.type_ids`.

```
locations.loc[(locations['type_ids'] == '527, 1, 14, 74, 120')]
```

	id	type_ids	lat	lng	unverified	description	season_start	season_stop	no_
547032	594876	527, 1, 14, 74, 120	53.487687	-3.000418	False	NaN	NaN	NaN	

The above is a record from `locations` with multiple values in the `type_ids` columns. When creating the join table, these cannot be processed into multiple records and they are dropped. The locations table will need to be cleaned to split out the `type_ids` column into multiple rows with a single `type_id`.

```
# Dropping any rows that have null type_ids, they're no use to us
locations = locations[locations['type_ids'].notna()]
# Converting type_ids to str guarantees no numeric values
locations['type_ids'] = locations['type_ids'].astype(str)
# Dropping any columns we're not interested in
locations = locations[['type_ids', 'lat', 'lng']]
```

```
/var/folders/xx/1jz4bzvs0kdghy416zlmdn5c0000gn/T/ipykernel_86879/3202321796.py:4: Setti
ngWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user
_guide/indexing.html#returning-a-view-versus-a-copy
  locations['type_ids'] = locations['type_ids'].astype(str)
```

```
temp_ids = []
processed_rows = []

for index, row in locations.iterrows():
    temp_ids = row.type_ids.split(", ")

    for temp_id in temp_ids:
        processed_rows.append([temp_id, row['lat'], row['lng']])
```

```
processed_rows[0:1]
```

```
[['3', 37.4098488210367, -122.137528895106]]
```

```
cleaned_locations = pd.DataFrame(processed_rows)
cleaned_locations.rename(columns={0: 'type_id', 1: 'lat', 2 : 'lng'}, inplace=True)
cleaned_locationds = cleaned_locations.notna()
cleaned_locations.head()
```

	type_id	lat	lng
0	3	37.409849	-122.137529
1	8	37.412087	-122.140182
2	4	37.412043	-122.139700
3	3	37.411562	-122.139288
4	4	37.411252	-122.138862

With the data cleaned up now, the join should be more successful. We'll convert `types.id` to string for comparison.

```
types['id'] = types['id'].astype(str)
cleaned_join = pd.merge(types, cleaned_locations, how='inner', left_on='id', right_on='
cleaned_join.shape[0]
```

```
1517708
```

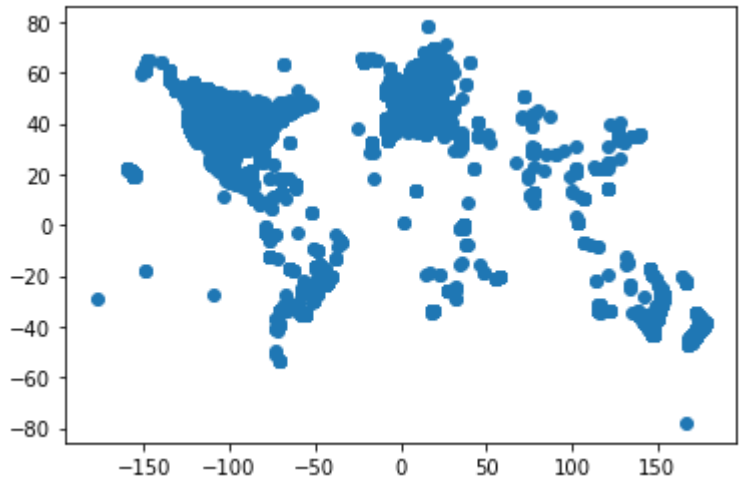
That's looking a lot healthier now. But there's still a lot of unused columns carried over from `types`. Let's get rid of those.

```
df = cleaned_join[['id', 'en_name', 'lat', 'lng']]
df.head()
```

	id	en_name	lat	lng
0	1	Plum	37.858421	-122.264114
1	1	Plum	37.870052	-122.297340
2	1	Plum	37.870041	-122.247513
3	1	Plum	37.843819	-122.277657
4	1	Plum	37.849087	-122.262802

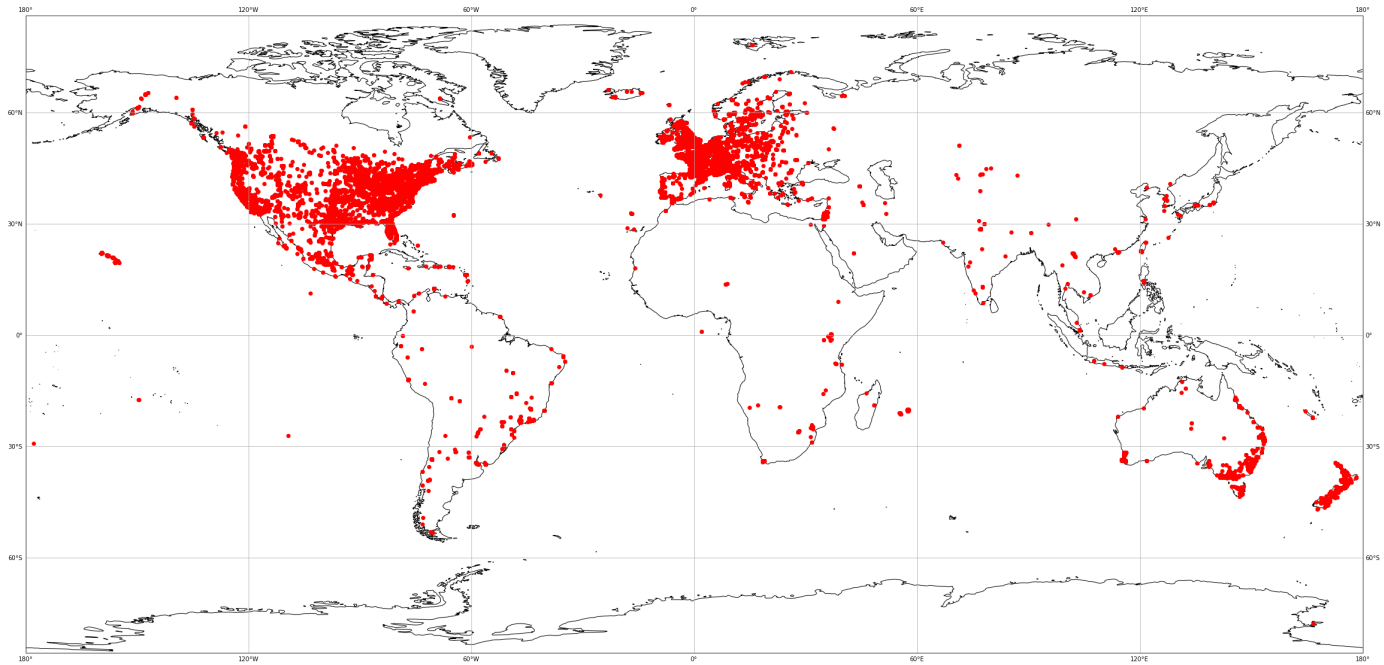
```
plt.scatter(df.lng, df.lat)
```

```
<matplotlib.collections.PathCollection at 0x7fe5d1bb4550>
```



Notice how the data point look reflective of a world map. This makes sense as they are lattitude and longitude plots. But we can see them on a world map below.

```
plt.figure(figsize=(40, 30))

ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines(resolution='50m')
ax.gridlines(draw_labels=True, dms=True, x_inline=False, y_inline=False)
plt.scatter(df['lng'], df['lat'], color='red')

plt.show()
```



Excellent. Now the data is clean and usable, let's have a go at clustering it ready for frequent pattern mining. Here we'll use K-Means clustering and arbitrarily use 1000 clusters so that we will have a good number of 'baskets' to use.

```
km = KMeans(n_clusters=1000)
y_predicted = km.fit_predict(df[['lat', 'lng']])
y_predicted
df.loc[:,('cluster')] = y_predicted
df.head()
```

id	en_name	lat	lng	cluster

	id	en_name	lat	lng	cluster
0	1	Plum	37.858421	-122.264114	546
1	1	Plum	37.870052	-122.297340	250
2	1	Plum	37.870041	-122.247513	250
3	1	Plum	37.843819	-122.277657	546
4	1	Plum	37.849087	-122.262802	546

We can use the `silhouette coefficient` to verify the quality of our clusters. Any value between 0 and 1 is okay, but closer to 1 is better.

```
X = df[['lng', 'lat']]
print(f'Silhouette Score (n=1000): {silhouette_score(X, km.labels_)}')
```

```
Silhouette Score (n=1000): 0.458763718782539
```

0.46, not bad, that's getting towards 75% unique. Ultimately, it's not too pressing that our clusters are unique. We are splitting data into baskets, not classifying new data. So long as our clusters are similar geographically (in terms of `lat` and `lng`, then we're fine.

Let's turn the clustered data in to baskets.

```
baskets = []

for i in pd.unique(df.cluster):
    line = ""

    for item in pd.unique(df[df['cluster']==i].en_name):
        line += f"{item},"

    baskets.append(line)

baskets[0:1]
```

```
["Plum,Dumpster (edible),Orange,Lemon,Avocado,Rosemary,Persimmon,Pomegranate,Apple,Pea
r,Grape,Lavender,Loquat,Kumquat,Common fig,Mandarin,Lime,Blackberry,Feijoa,Kale,Peach,N
ectarine,Prickly pear,Miner's lettuce,Olive,Strawberry tree,Tomato,Fennel,Green bean,Sq
uash,Nasturtium,Pumpkin,Almond,Quince,Walnut,Banana,Guava,Apricot,Globe artichoke,Mulbe
rry,Black mulberry,White mulberry,Black locust,Black walnut,Persian walnut,Cherry plum,
Kousa dogwood,Sour cherry,Sugar maple,Eastern redbud,Ginkgo,Hawthorn,Hackberry,Thyme,Sp
earmint,Borage,Common chickweed,Fruit tree,European beech,European linden,American pers
immon,Honey locust,Brazilian pepper tree,Citrus,Palm,Magenta lilly pilly,Black tupelo,B
ay laurel,Stone pine,Japanese flowering crabapple,Kurrajong,Queen palm,Holm oak,Carob,A
gave,Dogwood,California laurel,Japanese persimmon,European pear,Common hackberry,Southe
rn California walnut,Sydney golden wattle,Norfolk island pine,Pacific madrone,Guadalupe
palm,Paper mulberry,Jelly palm,Japanese camellia,Mediterranean hackberry,Netleaf hackbe
rry,Western redbud,Key lime,Cabbage tree,Common date palm,Mexican pinyon,Hollyleaf cher
ry,Black elderberry,Desert fan palm,Mexican fan palm,Honeysuckle,Japanese plum,Blue eld
erberry,European plum,Fig,Three-cornered leek,Purple deadnettle,English plantain,Pineap
pleweed,Shepherd's purse,Fruit,White clover,Scotch bonnet,Coast live oak,Kaffir lime,Du
mpster (non-edible),Marina strawberry tree,California buckeye,Broad-leaved paperpark,Co
mmon frangipani,Free library,Cow parsnip,Yellow-staining mushroom,California blackberr
y,Indian hawthorn,Garden nasturtium,Bigleaf maple,Chinese wisteria,Mission fig,Bavarian
fig,"]
```

At the end of each line, we are left with a `''` value. This can be stripped out later on easily. We'll convert the data to a frame, drop any null values, then convert it back to a list for analysing.

```
baskets = pd.DataFrame(baskets, columns=['food'])
baskets = baskets[baskets['food'].notna()]
```

```
data = list(baskets['food'].apply(lambda x:x.split(',') ))

for row in data:
    row.remove('')

data[0:1]
```

```
[['Plum',
  'Dumpster (edible)',
  'Orange',
  'Lemon',
  'Avocado',
  'Rosemary',
  'Persimmon',
  'Pomegranate',
  'Apple',
  'Pear',
  'Grape',
  'Lavender',
  'Loquat',
  'Kumquat',
  'Common fig',
  'Mandarin',
  'Lime',
  'Blackberry',
  'Feijoa',
  'Kale',
  'Peach',
  'Nectarine',
  'Prickly pear',
  "Miner's lettuce",
  'Olive',
  'Strawberry tree',
  'Tomato',
  'Fennel',
  'Green bean',
  'Squash',
  'Nasturtium',
  'Pumpkin',
  'Almond',
  'Quince',
  'Walnut',
  'Banana',
  'Guava',
  'Apricot',
  'Globe artichoke',
  'Mulberry',
  'Black mulberry',
  'White mulberry',
  'Black locust',
  'Black walnut',
  'Persian walnut',
  'Cherry plum',
  'Kousa dogwood',
  'Sour cherry',
  'Sugar maple',
  'Eastern redbud',
  'Ginkgo',
  'Hawthorn',
```

'Hackberry',
'Thyme',
'Spearmint',
'Borage',
'Common chickweed',
'Fruit tree',
'European beech',
'European linden',
'American persimmon',
'Honey locust',
'Brazilian pepper tree',
'Citrus',
'Palm',
'Magenta lilly pilly',
'Black tupelo',
'Bay laurel',
'Stone pine',
'Japanese flowering crabapple',
'Kurrajong',
'Queen palm',
'Holm oak',
'Carob',
'Agave',
'Dogwood',
'California laurel',
'Japanese persimmon',
'European pear',
'Common hackberry',
'Southern California walnut',
'Sydney golden wattle',
'Norfolk island pine',
'Pacific madrone',
'Guadalupe palm',
'Paper mulberry',
'Jelly palm',
'Japanese camellia',
'Mediterranean hackberry',
'Netleaf hackberry',
'Western redbud',
'Key lime',
'Cabbage tree',
'Common date palm',
'Mexican pinyon',
'Hollyleaf cherry',
'Black elderberry',
'Desert fan palm',
'Mexican fan palm',
'Honeysuckle',
'Japanese plum',
'Blue elderberry',
'European plum',
'Fig',
'Three-cornered leek',
'Purple deadnettle',
'English plantain',
'Pineappleweed',
"Shepherd's purse",
'Fruit',
'White clover',
'Scotch bonnet',
'Coast live oak',
'Kaffir lime',
'Dumpster (non-edible)',
'Marina strawberry tree',

```
 'California buckeye',
 'Broad-leaved paperpark',
 'Common frangipani',
 'Free library',
 'Cow parsnip',
 'Yellow-staining mushroom',
 'California blackberry',
 'Indian hawthorn',
 'Garden nasturtium',
 'Bigleaf maple',
 'Chinese wisteria',
 'Mission fig',
 'Bavarian fig']]
```

Now we have processed data rows ready for association rule mining.

Step 1 is to encode the data so that our `fpgrowth` algorithm can process it.

```
a = TransactionEncoder()
a_data = a.fit(data).transform(data)
tx = pd.DataFrame(a_data,columns=a.columns_)
tx = tx.replace(False,0)
tx.head()
```

	Abalone	Abelmoschus	Accolade flowering cherry	Acerola	Achacha	Achiote	Ackee	Acronychia	Adirondack crabapple	Afghan pine
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

5 rows × 2400 columns

Using 20% as our minimum support, we will get a returned dataframe containing any item that appears in more than 1:5 baskets.

```
fp = fpgrowth(tx, min_support=0.2, use_colnames=True)
fp.head()
```

	support	itemsets
0	0.574	(Apple)
1	0.399	(Blackberry)
2	0.384	(Dumpster (edible))
3	0.379	(Pear)
4	0.365	(Mulberry)

Next we run this through the `association_rules` function which will generate a list of rules meeting a minimum confidence threshold. Let's use `60%` .

```
fp_ar = association_rules(fp, metric = "confidence", min_threshold = 0.6)
```

```
fp_ar.head()
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	convicti
0	(Blackberry)	(Apple)	0.399	0.574	0.334	0.837093	1.458350	0.104974	2.6149
1	(Dumpster (edible))	(Apple)	0.384	0.574	0.278	0.723958	1.261251	0.057584	1.5432
2	(Apple)	(Pear)	0.574	0.379	0.350	0.609756	1.608855	0.132454	1.5913
3	(Pear)	(Apple)	0.379	0.574	0.350	0.923483	1.608855	0.132454	5.5673
4	(Blackberry)	(Pear)	0.399	0.379	0.244	0.611529	1.613533	0.092779	1.5985

Fantastic! We have rules. As a curisity, let's see how many rules our dataset produced.

```
fp_ar.shape[0]
```

```
52
```

52 rules. That's a lot. We have some high confidence rules in the `fp_ar.head()`. To get some really strong rules we can filter out rules that are below `80%` confidence.

```
fp_ar_strong = fp_ar.loc[(fp_ar['confidence'] >= 0.8)]
fp_ar_strong.shape[0]
```

```
17
```

Wow, 17. Still a bit high, let's try `90%`.

```
fp_ar_stronger = fp_ar.loc[(fp_ar['confidence'] >= 0.9)]
fp_ar_stronger.shape[0]
```

```
13
```

```
fp_ar_strongest = fp_ar.loc[(fp_ar['confidence'] >= 0.95)]
fp_ar_strongest.shape[0]
```

```
4
```

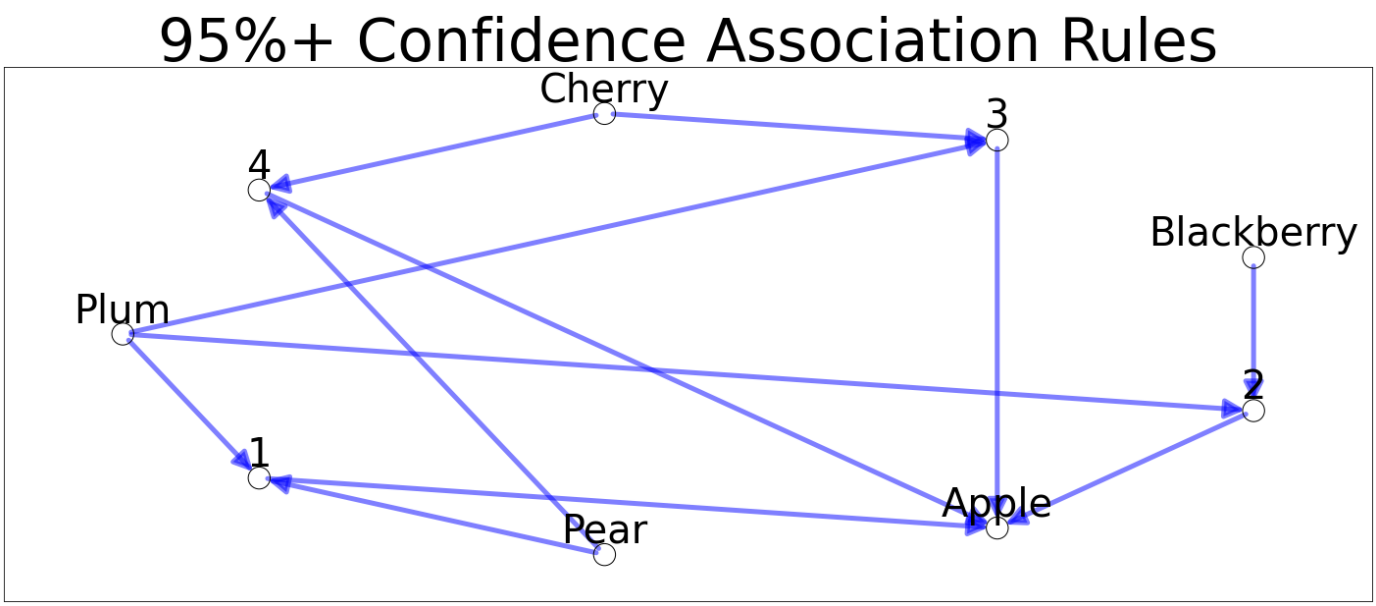
4 rules. That's more like it! All 4 of these rules have 95% confidence that `{X} => {Y}`.

```
fp_ar_strongest
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	convic
25	(Plum, Pear)	(Apple)	0.222	0.574	0.214	0.963964	1.679380	0.086572	11.821
29	(Plum, Blackberry)	(Apple)	0.215	0.574	0.205	0.953488	1.661130	0.081590	9.159
33	(Plum, Cherry)	(Apple)	0.217	0.574	0.210	0.967742	1.685962	0.085442	13.206
48	(Pear, Cherry)	(Apple)	0.239	0.574	0.232	0.970711	1.691135	0.094814	14.544

```
plt.figure(figsize=(25, 10))
```

```
plt.title('95%+ Confidence Association Rules', fontsize=60)
found_edges = []
count = 1
G = nx.DiGraph()

for index, row in fp_ar_strongest.iterrows():
    for item in row['antecedents']:
        found_edges.append((item, count))
    for item in row['consequents']:
        found_edges.append((count, item))
    count += 1

G.add_edges_from(found_edges)
pos = nx.shell_layout(G)
pos_higher = {}
y_off = 0.1  # offset on the y axis

for k, v in pos.items():
    pos_higher[k] = (v[0], v[1]+y_off)

nx.draw_networkx_nodes(G, pos, node_size=500, node_color='none', edgecolors='black')
nx.draw_networkx_edges(G, pos, edgelist=G.edges(), edge_color='blue', arrows=True, widt
nx.draw_networkx_labels(G, pos_higher, font_color='black', font_size=40)

plt.show()
```



Some Useful Links

https://medium.com/@mervetorkan/association-rules-with-python-9158974e761a

http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/fpgrowth/#frequent-itemsets-via-the-fp-growth-algorithm

https://www.geeksforgeeks.org/directed-graphs-multigraphs-and-visualization-in-networkx/

https://towardsdatascience.com/silhouette-coefficient-validating-clustering-techniques-e976bb81d10c#:~:text=Silhouette%20Coefficient%20or%20silhouette%20score%20is%20a%20metric%20